IRsmallDecoder

A small, fast and reliable infrared signals decoder, for controlling Arduino projects with remote controls.

This is a Library for receiving and decoding IR signals from remote controls. Perfect for your Arduino projects that need a fast, simple and reliable decoder, but don't require the usage of multiple different protocols at the same time and don't need to send IR signals.

Table of Contents

Main features

- The signals are fully decoded and the data is divided into separate variables;
- The initial repetition codes are ignored, effectively reducing the possibility of getting several codes when we just want one;
- Held Keys (or buttons) are detected and processed in a more useful way;
- The signal's tolerances are very loose, allowing a high detection rate without compromising the reliability;
- The signal's redundant data is only used for error detection (if the protocol has it);
- SRAM and Flash memory usage is very low;
- The decoding is done asynchronously, no timers required, so you can use them for other things;
- No conflicts with timer-related functionalities such as *tone()*, servos, *analogWrite()*, etc.;

- The signal acquisition and processing is done by a hardware (external) interrupt;
- This library is compatible with a wide range of the Arduino boards.

Supported protocols

- NEC
- NECx
- Philips RC5 and RC5x (simultaneously)
- Sony SIRC 12, 15 and 20 bits (individually or simultaneously)
- SAMSUNG old standard
- SAMSUNG 32 bits (16 of which are for error detection)

Supported boards

Because no hardware specific instructions are used, it probably works on all Arduino boards (and possibly others, I'm not quite sure, I've only tested it thoroughly on an Arduino Uno and a Mega).

ATtiny 25/45/85/24/44/84 microcontrollers are supported.

If you have problems with this library on some board, please submit an issue here or contact me.

Connecting the IR receiver

The receiver's output must be connected to one of the Arduino's digital pin that is usable for interrupts and it also must work with the CHANGE mode if the intended protocol uses that mode. One example of a board that does not have CHANGE mode on some of the interrupt pins is the Arduino 101 and one protocol that uses that mode is the RC5 (you can find the modes here).

The following table (adapted from the Arduino Reference) contains the digital pins that can be use to connect the IR receiver to the Arduino board:

Board or microcontroller	Digital pins usable for interrupts
Uno, Nano, Mini, other 328-based	2, 3
Uno WiFi Rev.2, Nano Every	all digital pins
Mega, Mega2560, MegaADK	2, 3, 18, 19, 20, 21*
Micro, Leonardo, other 32u4-based	0, 1, 2, 3, 7
Zero	all digital pins, except 4
MKR Family boards	0, 1, 4, 5, 6, 7, 8, 9, A1, A2
Nano 33 IoT	2, 3, 9, 10, 11, 13, A1, A5, A7
Nano 33 BLE, Nano 33 BLE Sense	all pins
Due	all digital pins
101 with CHANGE mode	2, 5, 7, 8, 10, 11, 12, 13
101 with other modes	all digital pins

Board or microcontroller	Digital pins usable for interrupts
ATtiny 25/45/85	2**
ATtiny 24/44/84	8**

> [*] - In the Mega family, pins 20 & 21 are not available to use for interrupts while they are used for I2C communication.
>
> [**] - Assuming you're using damellis' ATtiny core or SpenceKonde's ATTinyCore. Other cores may have different pin assignments.

If you're not sure about how to connect the IR receiver to the Arduino, go to: IR receiver connection details at the end of this document.

Installing the library

With the Library Manager

- Run Arduino IDE and go to *tools* > *Manage Libraries...*
  or *Sketch* > *Include Library* > *Manage Libraries...*;
- Search for IRsmallDecoder and install.

Manually

- Navigate to the Releases page;
- Download the latest release (zip file);
- Run Arduino IDE and go to *Sketch* > *Include Library* > *Add .ZIP Library*;
- Or, instead of using Arduino IDE, extract the zip file and move the extracted folder to your libraries directory.

Using the library

In the INO file, **one** of the following directives must be used:

```
#define IR_SMALLD_NEC
#define IR_SMALLD_NECx
#define IR_SMALLD_RC5
#define IR_SMALLD_SIRC12
#define IR_SMALLD_SIRC15
#define IR_SMALLD_SIRC20
#define IR_SMALLD_SIRC
#define IR_SMALLD_SAMSUNG
#define IR_SMALLD_SAMSUNG32
```

before the

```
#include <IRsmallDecoder.h>
```

Then you need to create **one** decoder object with the correct digital pin:

```
IRsmallDecoder irDecoder(2); //IR receiver connected to pin 2 in this example
```

And also a decoder data structure:

```
irSmallD_t irData;
```

Inside the loop(), check if the decoder has new data available. If so, do something with it:

```
void loop() {
  if(irDecoder.dataAvailable(irData)) {
    Serial.println(irData.cmd, HEX);
  }
}
```

A full example

```
#define IR_SMALLD_NEC
#include <IRsmallDecoder.h>
IRsmallDecoder irDecoder(2);
irSmallD_t irData;

void setup() {
  Serial.begin(250000);
  Serial.println("Waiting for a NEC remote control IR signal...");
  Serial.println("held \t addr \t cmd");
}

void loop() {
  if(irDecoder.dataAvailable(irData)) {
    Serial.print(irData.keyHeld,HEX);
    Serial.print("\t ");
    Serial.print(irData.addr,HEX);
    Serial.print("\t ");
    Serial.println(irData.cmd,HEX);
  }
}
```

Address check

if you are using multiple remotes, with different addresses, in the vicinity of your project, you should also verify the address. You can do something like this:

```
if (irData.addr == theRightAddr) {
   switch (irData.cmd) {
      case someCmd:
        // do something here
        break;
      case someOtherCmd:
        // do some other things
        break;
      //etc.
     }
}
```

The multifunctional *dataAvailable()* method

The `dataAvailable(irData)` method combines the functionalities of 3 "fictitious" functions: *isDataAvailable()*, *getData()* and *setDataUnavailable()*. If there is some data available, already decoded, when `irDecoder.dataAvailable(irData)` is called:

- The data is copied to the selected data structure `irData`;
- The original data is marked as unavailable;
- And, finally, it returns true.

If there's no new data, it just returns false.

Note: this library does not use data buffering, if a new signal is received before the available data is retrieved, that previous data is discarded. This may happen if the loop takes to long to check for new data. So, if you want to use repetition codes, try to keep the loop duration below 100ms (for NEC and RC5) and don't use delays. They don't interfere with the decoding, but I don't recommend their usage.

If you want to check if any key was pressed and don't care about the data, you can use the `dataAvailable()` method without any parameters. Keep in mind that, if there's new data available, this method will discard that data, before returning true. The ToggleLED example demonstrates this functionality.

Disabling the decoder

If you don't want to receive IR codes when they are not needed or if you want to prevent possible interferences in time critical functions by the decoders' interrupts, you can use the `disable()` and `enable()` methods. The TemporaryDisable example demonstrates a possible usage for these methods.

The `enable()` method also resets the decoder after reenabling it. This is useful if you have to temporarily disable all interrupts or need to use a library that does that. If you don't reset the decoder after re-enabling all interrupts, the next IR signal may not be recognized if the IR receiver detected some signal while the interrupts where disabled.

Protocol data structures

The protocol data structure is not the same for all protocols, but they all have two member variables in common:

- **cmd** - the button command code (one byte);
- **addr** - the address code (usually the same for all buttons on one single remote).

Most of the decoders have the **keyHeld** variable (which is set to *true* when a button is being held) and two of the SIRC decoders have the **ext** variable (see notes for more details);

The following table shows the number of bits used by each protocol and the datatypes of the data structure member variables:

Protocol	keyHeld	cmd	addr	ext
NEC	bool	8/uint8_t	8/uint8_t	--
NECx	bool	8/uint8_t	16/uint16_t	--
RC5	bool	7/uint8_t	5/uint8_t	--
SIRC12	--	7/uint8_t	5/uint8_t	--
SIRC15	--	7/uint8_t	8/uint8_t	--
SIRC20	--	7/uint8_t	5/uint8_t	8/uint8_t
SIRC	bool	7/uint8_t	8/uint8_t	8/uint8_t
SAMSUNG	bool	8/uint8_t	12/uint16_t	--
SAMSUNG32	bool	8/uint8_t	8/uint8_t	--

Notes

- Only one protocol can be compiled at a time, however:
  - NECx also decodes NEC, but the address will have redundant data;
  - The RC5 implementation also decodes the extended protocol version, which has a field bit that is used as an extra command bit (making a total of 7 bits);
  - SIRC12 will detect signals from SIRC15 and SIRC20, but the codes will not be correct;
  - Similarly, SIRC15 will also detect signals from SIRC20, but not from SIRC12.
- SIRC handles 12, 15 and 20 bits at the same time, by taking advantage of the fact that most Sony remotes send three signal frames each time one button is pressed. It uses triple frame verification, checks if a key was held and ignores the initial repetition codes.
- SIRC12, SIRC15 and SIRC20 use a basic (smaller and faster) implementation, without the triple frame verification and without the **keyHeld** check.
- The SIRC20 protocol has the **ext** variable which holds extended data.
- The SIRC decoder also has the **ext** variable, but it's only used when a 20-bit code is detected, otherwise it's set to 0.

Possible improvements

- I might add a few more IR protocols to this library (there are a lot of them out there);
- The keyHeld initial delay is hard-coded, I could make it configurable (in constructor) or even changeable (with method);
- I believe it may be possible to increase the number of usable pins, by using NicoHood's PinChangeInterrupt Library;
- SIRC12, SIRC15 and SIRC20 do not have the keyHeld feature. SIRC fills that gap, but requires 3 signal frames for each keypress;
- The SIRC decoder could also return the number of detected bits (12, 15 or 20).
- I have not finished redrawing the IR signals' graphs that would better explain the protocols' timings.

Contributions

So far, these releases were made without any significant contribution from other developers, but I do have to say that this work was inspired by some of the existing IR Libraries: Arduino-IRremote, IRLib2, IRReadOnlyRemote, Infrared4Arduino and especially the IRLremote, which was almost what I was looking for, but not quite... So I decided to make my own NEC decoder and then an RC5 and a SIRC. Finally I decided to put these decoders in a library, hoping that it will be useful to someone.

Contact information

If you wish to report an issue related to this library (and don't want to do it on GitHub) you may send an e-mail to: lumica@outlook.com. Suggestions and comments are also welcome.

License

Appendix A - Details about this library

Size

The size of this library is, as the name implies, small (about 900 bytes on average, for the Arduino UNO board) and the memory usage is also reduced (around 30 bytes). Keep in mind that these values vary depending on the selected protocol and the board used.

Program memory and static data used (in SRAM) on an Arduino UNO (in bytes):

Protocol	Program memory	Static data
NEC	862	29
NECx	858	31
RC5	1066	32
SIRC12	710	23
SIRC15	686	23
SIRC20	768	27
SIRC	1266	38
SAMSUNG	884	30
SAMSUNG32	856	30

To keep track of the sizes of this library, I used a sketch, similar to the ToggleLED example, with and without the library. By compiling each of the supported protocols and comparing their sizes with the reference sketch we get the memory used by the library.

Reference sketch

```
// #define IR_SMALLD_NEC
// #include <IRsmallDecoder.h>
// IRsmallDecoder irDecoder(2);
// irSmallD_t irData;
int ledState=LOW;
void setup() {
  pinMode(LED_BUILTIN, OUTPUT);
}
void loop() {
//if(irDecoder.dataAvailable(irData)){
    ledState=(ledState==LOW)? HIGH:LOW;
    digitalWrite(LED_BUILTIN,ledState);
//}
}

// On Arduino UNO,
// Sketch uses 766 bytes
// Global variables use 11 bytes
```

With NEC protocol decoder

```
#define IR_SMALLD_NEC
#include <IRsmallDecoder.h>
IRsmallDecoder irDecoder(2);
irSmallD_t irData;
int ledState=LOW;
void setup() {
  pinMode(LED_BUILTIN, OUTPUT);
}
void loop() {
  if(irDecoder.dataAvailable(irData)){
    ledState=(ledState==LOW)? HIGH:LOW;
    digitalWrite(LED_BUILTIN,ledState);
  }
}

// On Arduino UNO, with NEC protocol,
// Sketch uses 1628 bytes
// Global variables use 40 bytes
```

Speed

Although my main goals are functionality and small size, I believe this library is reasonably fast. I haven't compared it to other libraries (it's not easy to do so), but I was able to compare the speed of the different protocols that I've implemented so far:

Protocol Speed comparisons:

Protocol	Interrupt Mode	Avg. Interrupt Time	Max. Interrupt Time	Interrupts per Keypress	Signal Duration
NEC	FALLING	11.4 µs	13 µs	34	67.5ms
NECx	FALLING	10.9 µs	13 µs	34	67.5ms
RC5	CHANGE	10.4 µs	17 µs	14 to 28	24.9ms
SIRC12	RISING	10.3 µs	13 µs	3*13	3*(17.4 to 24.6)ms
SIRC15	RISING	10.5 µs	12 µs	3*16	3*(21 to 30)ms
SIRC20	RISING	11.1 µs	15 µs	3*21	3*(27 to 39)ms
SIRC	RISING	11.7 µs	17 µs	39, 48 or 63	3*(17.4 to 39)ms
SAMSUNG	FALLING	11.0 µs	13 µs	2*22	2*(32.1 to 54.6)ms
SAMSUNG32	FALLING	11.0 µs	14 µs	34	(54.6 to 72.6)ms

Notes:

- Signal Duration is the effective signal duration, not the signal period;
- Tested on an Arduino Uno @ 16MHz;
- To get the number of the clock cycles used by an interrupt, multiply the time (in µs) by 16;
- The decoding is done partially while the signal is being received. When a signal is fully received, the final stage of the decoding is executed and that's when the interrupt takes more time to run.

Unwanted initial repetition codes

Remote control keys do not "bounce", but the remotes do tend to send more codes than we wish for when we press a button. That's because, after a very short interval, they start sending repeat codes. To avoid those unwanted initial repetitions, this library ignores a few of those repetition codes before confirming that the button is actually being held.

Data separation

The data sent by the remotes is decoded according to the protocols' specifications and separated into different variables. On most remotes only the 8-bit command matters, so you don't have to work with 16-bit or 32-bit codes, reducing code size and memory usage.

Simplicity

As you've probably seen above, or if you've already tried one of the "Hello..." examples, this library is very simple to use and it's not full of rarely needed features. That's what makes it small and why it uses few

resources. Additional features may be added in the future, but only if requested and do not significantly affect the size and/or speed.

How it works

The decoding is done asynchronously, which means that it doesn't rely on a timer to receive and process signals, but it uses a hardware interrupt to drive the Finite State Machines that perform the decoding. In fact, they are Statechart Machines (David Harel type) working in a asynchronous mode.

Most of the protocols' Statechart Machines are implemented using *switch cases*, but I also use the "labels as values" GCC extension (AKA "computed gotos") to implement some of the more complex statecharts. It's not a C++ standard but it should work with all IDEs that use the GCC (like Arduino IDE). If you have problems compiling any of the protocols that use the "labels as values" extension, please submit an issue here or contact me.

I can't say that it's easy to understand how these decoders work, some of the Statechart Machines I designed turned out to be a bit tricky. But if you still want to take a look at the statechart diagrams, they can be found here. Note that they may not be an exact representation of what I actually implemented, but they are a good starting point.

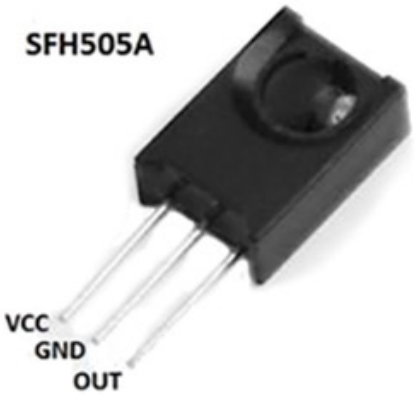
No hardware specific instructions

In order to make this library compatible with most of the Arduino boards, I didn't include any hardware specific instructions, but I did use a programming technique in which it's assumed that the microcontroller's endianness is Little-Endian. On some boards you may even get a warning related to this, but it should work anyway.

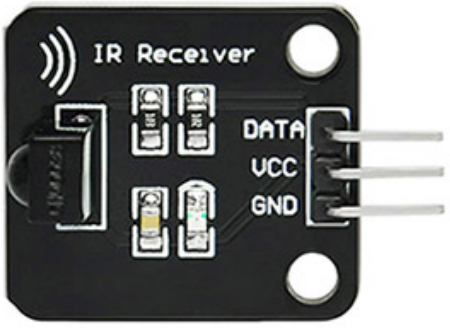
Appendix B - IR receiver connection details

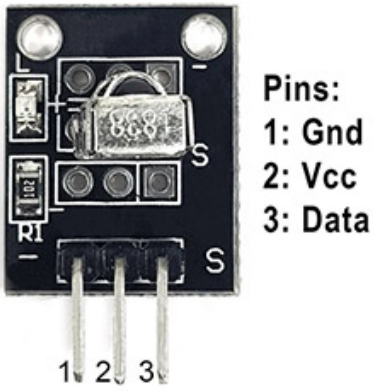
If you are using a simple IR receiver module, the pinout order will most likely be `Out Gnd Vcc`, as in the following examples:

But beware, there are other IR receivers with different pinouts, like these examples:



If you are using a test module, the pinout is usually written on it (sometimes it's DATA, DAT or S instead of OUT).



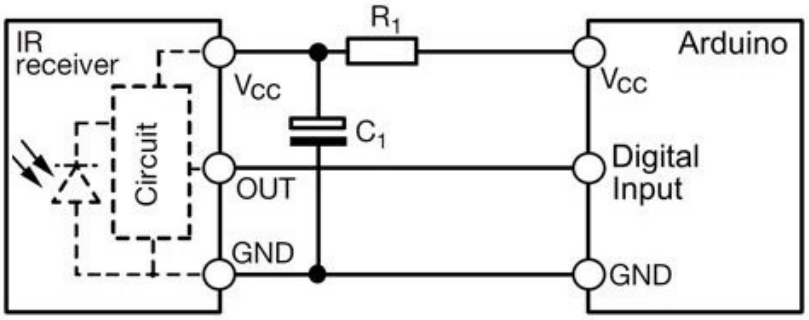


The connection to the Arduino is very straightforward, just connect:

- OUT (or DAT or S) to one of the Arduino's digital pin that has interrupt capability[1];
- VCC to the Arduino's +5V or (+3.3V if you are using a board with a lower operating voltage[2]);
- GND to one of the Arduino's Ground connector.

> 1. Go to Connecting the IR receiver for more information.
> 2. Keep in mind that not all IR receivers can operate at low voltages.

Nearly all IR receiver's datasheets recommend the usage of an RC filter (R1, C1) at the power input, but it's not absolutely necessary, (it's meant to suppress power supply disturbances):



Note that most IR receiver test modules already have that RC filter.